Chapter 1
Architecture specifications in CλaSH

Jan Kuper, Christiaan Baaij, Matthijs Kooijman, Marco Gerards
University of Twente, Dept. of Comp. Science, Enschede, The Netherlands
Email: j.kuper@utwente.nl

Abstract

This paper introduces CλaSH, a novel hardware specification environment, by discussing several non-trivial examples. CλaSH is based on the functional language Haskell, and exploits many of its powerful abstraction mechanisms such as higher order functions, polymorphism, lambda abstraction, pattern matching, type derivation. As a result, specifications in CλaSH are concise and semantically clear, and simulations can be directly executed within a Haskell evaluation environment. CλaSH generates synthesizable low-level VHDL code by applying several transformation rules to a functional specification of a digital circuit.

1.1 Introduction

A synchronous combinational digital circuit (without feedback) transforms input signals into output signals. Each time such a circuit gets the same input signals, it produces the same output signals, i.e., it behaves as a mathematical function. Things become a bit more complicated when a circuit contains memory elements, i.e., when the circuit has state, since a (mathematical) function does not have state. Still, intuitively a circuit strongly refers to the concept of function and several attempts have been made to develop hardware description languages based on a functional language, see e.g. [2], [5]–[8].

Two of the most well-known of these are *Lava* (see [2]) and *ForSyDe* (see [6]). These languages are domain specific embedded languages, and are both defined in Haskell. In both languages a digital circuit is specified as a function which operates on (possibly infinite) *streams* of values, where at the same time a clock is represented in the stream: at each clock cycle one stream element is processed. Furthermore, both Lava and ForSyDe model state by a

delay function which intuitively holds each stream element during one clock cycle.

In CλaSH we take a different perspective. Instead of defining a domain specific embedded language, CλaSH compiles specifications written in (an extended subset of) plain Haskell itself. Furthermore, these specifications do not work explicitly on streams of signals, but rather express a *structural* description of a circuit. In order to model state, CλaSH considers a circuit as a Mealy Machine, i.e., the function representing the behaviour of a circuit with state has *two* argument: the (current) state s and the (tuple of) input signal(s) i. The result of the function also consists of two things: the (new) state s' and the output signal(s) o, i.e., the result is of the form (s', o). Thus, CλaSH assumes that the type of a function *arch* describing a hardware architecture, i.e., the type of a circuit specification, is as follows:

$$arch :: State \to Input \to (State, Output)$$

for appropriate types *State*, *Input*, *Output*. Note that the function *arch* is a binary function which is first applied to a state s and only then to an input i. Thus, an application of *arch* to its arguments s and i is written as

$$arch\ s\ i$$

and not as

$$arch\ (s, i)$$

which would be a more generally known form of an application of a binary function *arch*. Though the second form is possible in CλaSH, the first way of writing is advantageous for e.g. partial application. We will see an example of partial application in section 1.3.3.

A second difference between the aforementioned languages and the method described in this paper is how the clock is dealt with. For CλaSH the clock is not explicitly expressed, instead it is assumed that a specification describes the functionality performed during one clock cycle.

A third simplification in comparison to other functional HDL's is found in the way how to *simulate* a given specification. Since CλaSH specifications are written in Haskell itself, simulation comes more or less for free. We only need a function *simulate*, which is the same for every architecture specification of the type of *arch* above. It is recursively defined as follows:

$$\begin{aligned} &simulate\ f\ s\ (i : is) = o : simulate\ f\ s'\ is \\ &\quad \textbf{where} \\ &\qquad (s', o) = f\ s\ i \end{aligned}$$

In this definition, the argument f is the function that specifies a circuit, s is the state, and $i : is$ is the stream of input signals, with i the first input

signal, and is the remaining stream of the input signals. In the *where* clause the function f is applied to the state s and the first input signal i, which results in the new state s' and the output signal o.

Then the output stream consists of the output o, followed by the result of the *simulate* function applied to the same hardware specification f, the new state s' and the remaining stream is of input signals. As mentioned before, this approach expresses a *Mealy machine* (see Figure 1.1). Note that the function *simulatie* is a *higher order function* because its first argument f is a function itself.

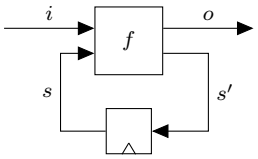

Fig. 1.1 Mealy machine

As a final feature of our approach we mention that several abstraction mechanisms are automatically available, such as choice mechanisms, higher order functions, polymorphism, lambda abstraction, and derivability of types.

On the other hand, some features of Haskell, such as dynamic data structures (lists, trees) and unlimited recursion do not have a direct counterpart in hardware. However, when at compile time the maximum size of data structures, or the maximum number of recursions is known, hardware can in principle be generated. In future, CλaSH will be extended with these possibilities.

The focus of this paper is to introduce CλaSH by discussing several examples, each illustrating some specific language constructs (section 1.3). The examples are preceded by a description of a few special types and operations that are needed for hardware descriptions (section 1.2). Because of lack of space detailed evaluations fall outside the scope of this paper.

1.2 Preliminary remarks

In CλaSH the following constructions that are typically needed for hardware specifications are pre-defined.

Hardware types

The two most elementary types are the types *Bit* and *Bool*. The first type contains the values *Low* and *High*, the second the values *True* and *False*.

For integers the constructor *Signed* is available to indicate the number of bits involved, as in: *Signed* 16, *Signed* 32, etc. There also is the constructor *Index*: the type *Index* 12 means that the integer values of this type fall in the reange $0 \cdots 12$.

CλaSH recognizes vector types: *Vector* n a, where n is an integer (typically of *Index*-type) and a an already given type[1]. Naturally, this type denotes a vector of n elements (with indexes $0 \cdots n-1$) of type a. Assuming numbers of type *Signed* 16, the expression $V\ [1, 2, 3, 4]$ is an example of a value of type *Vector* 4 (*Signed* 16).

User defined types

The designer can also define his own types, though in the present prototype of CλaSH that possibility is limited to some special cases of so-called "algebraic types". We will discuss examples of this in Section 1.3.

Operations and functions

In CλaSH several standard Haskell functions for lists have been redefined for vectors. For example, the function *init* removes the last element of a vector, whereas *last* returns the last element of a vector.

The operation $\triangleright$ adds an element in front of a vector, and $\triangleleft$ adds an element to the end of a vector. Likewise, the operations $\bowtie$, $\Lsh$ shift an element into a vector from the left, right, respectively, and move the other elements one position to the right, left, respectively. Thus, where $x \triangleright xs$ is one element longer than the original vector xs, $x \bowtie xs$ has the same length as xs.

Higher order functions such as *map*, *zipWith*, etc, which are standard in Haskell, are redefined for vectors as well.

Compilation pipeline

The focus of this paper is on showing the *usage* of CλaSH in a series of examples, but a few words on the compilation pipeline according to which CλaSH proceeds are in place. During this compilation pipeline a CλaSH specification is transformed in a number of steps into synthesizable VHDL.

The first step is performed by the Haskell compiler GHC which translates the CλaSH specification into an intermediate language, called *Core*. This result is then transformed by applying a set of *rewrite rules* into a *normal form*, which is close to VHDL. The final step, translation of this normal

[1] The notation *Vector* n a is a slightly simplified version of the notation used in CλaSH, but that does not influence the rest of this paper.

form into VHDL, is now relatively simple (for details, see [1, 4]). In fact, the rewriting process results in a *Core* expression that is very close to a netlist format. The reason to choose for a translation into VHDL is the availability of a well-developed toolchain for VHDL simulation and synthesis.

1.3 Examples

In section 1.3.1 we discuss a simple multiply-accumulate architecture, in section 1.3.2 some variants of a fir filter are shown, in section 1.3.3 a simple cpu, in section 1.3.4 a floating point reduction circuit.

1.3.1 Multiply-accumulate

The first example is a simple multiply-accumulate function *mac* (see Figure 1.2). The input consists of a sequence of pairs of integer numbers (x, y) that have to be pairwise multiplied and accumulated in the state s, which in this case consists of a single integer number:

$$
\begin{aligned}
&mac\ acc\ (x, y) = (acc', acc') \\
&\quad \textbf{where} \\
&\qquad acc' = acc + x * y
\end{aligned}
$$

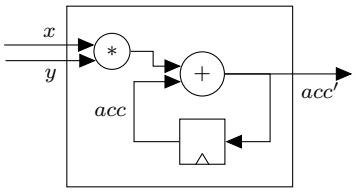

Fig. 1.2 Multiply-accumulate

The following is an example of a simulation[2]:

$$simulate\ mac\ 0\ \langle (1,2),(3,4),(5,6)\rangle\ =\ \langle 2, 14, 44\rangle$$

[2] Actually, to let the simulation in Haskell end properly, the definition of *simulate* above has to be extended with a clause for the empty input sequence in case the total input sequence is finite.

Note that in the specification of *mac* above there is some polymorphism present: it works for any type of value for which + and $*$ exist. So, before CλaSH can translate this definition into synthesizable VHDL, we have to fix the type of *mac*. For example, we might define the type for *mac* as follows:

$$mac :: Signed\ 16 \to (Signed\ 16, Signed\ 16) \to (Signed\ 16, Signed\ 16)$$

i.e., the first argument (the state) is of type $Signed\ 16$ and the second argument (the input) is a pair of type $(Signed\ 16, Signed\ 16)$. The result again is a pair of type $(Signed\ 16, Signed\ 16)$, of which the first is the new state, and the second one is the output. That is to say, all values are integers of 16 bits long.

Remarks

This first example requires no special definitions or functions and the correspondence between the specification of *mac* and Figure 1.2 is immediate.

1.3.2 Variants of a fir-filter

A finite impulse response (fir) filter calculates the dot product of two vectors, i.e., it pairwise multiplies a vector of fixed constants (h_i) with an equally long substream of the input (x_t), and then adds the results. Thus, the result y_t of a fir-filter at time t is defined as follows:

$$y_t = \sum_{i=0}^{n-1} x_{t-i} * h_i \tag{1.1}$$

There are many implementations of a fir filter, we show three of them to illustrate that their differences can be concisely expressed in the CλaSH definitions. In the context of this paper we assume that every clock cycle a new input value arrives.

Variant 1

An Haskell definition which is equivalent with equation 1.1 is as follows (*hs* is the vector of constants, *xs* is the substream of inputs, • stands for the dot product of two vectors):

$$xs \bullet hs = foldr\ (+)\ 0\ (zipWith\ (*)\ xs\ hs)$$

The function $zipWith$ is a standard Haskell function which pairwise applies a binary operation (here: multiplication) to the elements of two vectors (here: xs and hs).

The functions $foldl$, $foldr$ are standard Haskell functions which accumulates the elements in a vector by applying a binary operation to them (here: addition), starting from an initial value (here: 0). Note that $foldl$ accumulates from left to right, whereas $foldr$ accumulates from right to left, i.e., in backward order through a vector.

The functions $foldl$, $foldr$, $zipWith$ are higher order functions since they take a binary operation as their first argument.

The direct implementation fir_1 is now specified in Haskell as follows (see Figure 1.3):

$$
\begin{aligned}
&fir_1\ (hs, us)\ x = (\ (hs, us')\ ,\ y\ ) \\
&\quad \textbf{where} \\
&\qquad us' = x \rhd\!\!\rhd us \\
&\qquad y \quad = us \bullet hs
\end{aligned}
$$

Thus, the state of the function fir_1 is a pair of two vectors: the fixed values hs, and the vector us of stored input values that have to be kept in a sequence of registers. Note that $u_i = x_{t-i}$ and that the vector $hs = \langle h_3, h_2, h_1, h_0 \rangle$ in order to match the indexing of the h-values in definition 1.1.

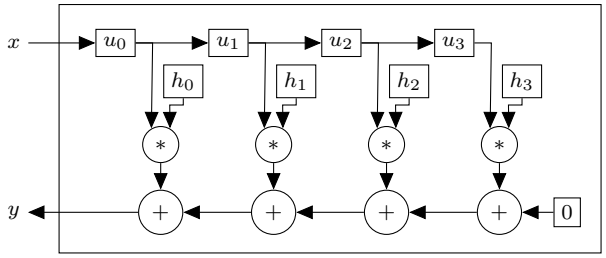

Fig. 1.3 fir-filter, variant 1

The result of fir_1 consists of two things. First, it contains the new state us' which is created from the odl state us by shifting the input value x in at the left (and thus discarding the "oldest" value in us). The hs-part of the state remains unchanged. The second part of the result is the output value y, i.e., the dot product of the full sequence us and hs.

Clearly, the first register u_0 may be left out. In that case the output would be

$$y = (x \rhd us) \bullet hs.$$

Also, the explicit mentioning of the initial value 0 is somewhat redundant. By defining

$$xs \bullet hs = foldr1\ (+)\ (zipWith\ (*)\ xs\ hs)$$

the accumulation would start by adding the last two elements and then proceeding as before.

Variant 2

An alternative definition fir_2 of a fir-filter is shown in Figure 1.4 and defined as follows:

$$\begin{aligned} &fir_2\ (hs, vs)\ x = (\ (hs, tail\ vs')\ ,\ head\ vs'\ ) \\ &\quad \textbf{where} \\ &\qquad ws = map\ (\backslash h \to h * x)\ hs \\ &\qquad vs' = zipWith\ (+)\ (vs \triangleleft x)\ ws \end{aligned}$$

The standard Haskell function *map* applies a function to all elements of a vector. In this case that function is denoted by a lambda term which expresses that the argument h is multiplied with x. Thus, by using *map*, all elements in hs are multiplied with x. Next, the results of this are pairwise added to the values in $0 +> vs$, i.e., a zero prefixed to vs.

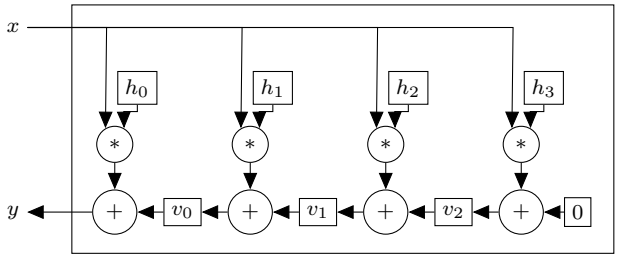

Fig. 1.4 fir-filter, variant 2

Variant 3

Finally, a third definition fir_3 goes as follows (see Figure 1.5):

$$
\begin{aligned}
&\mathit{fir}_3\ (hs, us, vs)\ x = ((hs, tailus < +x, init\ vs'), last\ vs') \\
&\quad \textbf{where} \\
&\qquad ws = zipWith\ (*)\ hs\ (us < +x) \\
&\qquad vs' = zipWith\ (+)\ (0+ > vs)\ ws
\end{aligned}
$$

It should be clear by now how the $zipWith$ functions take care of the pairwise multiplication and addition. Note that with this last definition the input value x should arrive every other clock cycle, and only every other clock cycle a valid result is delivered.

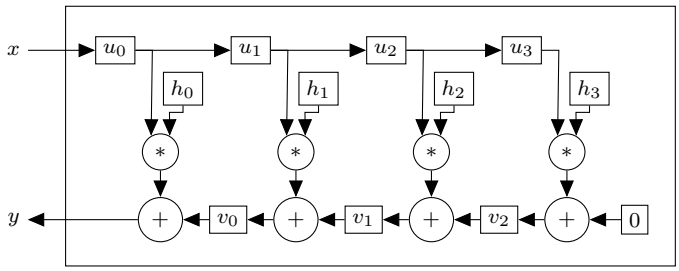

Fig. 1.5 fir-filter, variant 3

Remarks

The variants of the fir-filters above exploit several standard higher order functions (map, $zipWith$, $foldl1$) which are translated by CλaSH to synthesizable VHDL. Also λ-abstraction is recognized by CλaSH, as can be seen in variant 2.

These features give a high abstraction level to the designs of the fir-filters which makes the essential differences between these variants immediately visible and analyzable, as a comparison of the above definitions shows.

Clearly, as with the multiply-accumulate example, the polymorphic character of these functions leave the concrete type of the fir-filters undecided, so in order to specify concrete hardware, one still has to decide on the types of the fir-filters. The types of fir_1, fir_2, fir_3 differ slightly, for example, the state of fir_3 is a tuple of three vectors, whereas for fir_1, fir_2 the state is a tuple of two vectors. However, the pattern of the type definitions is the same for all three variants, and coincides with the pattern of the general type of the function $arch$ as shown in Section 1.1.

Finally, note that the above definitions hold for any number of taps in the fir-filters. This number is fully determined by the *Vector* type for the state parameters chosen by the designer.

1.3.3 Higher order cpu

Next, we describe a higher order cpu, containing three function units $fun\ 0$, $fun\ 1$, $fun\ 2$ (see Figure 1.6) each of which can perform a binary operation. Every function unit has six data inputs (of type $Signed\ 16$), and two address inputs (of type $Index\ 5$) that indicate which of the six data inputs are to be used as operands for the binary operation that the function unit performs. These six data inputs consist of one external input x, two fixed initialization values (0 and 1), and the previous output of each of the three function units. The output of the cpu as a whole is the previous output of $fun\ 2$. Function units $fun\ 1$ and $fun\ 2$ can perform a fixed binary operation, whereas $fun\ 0$ has an additional input for an opcode to choose a binary operation out of a few possibilities. Each function unit outputs its result into a register, i.e., the state of the cpu is a vector of three $Signed\ 16$ values:

$$\textbf{type}\ CpuState\ =\ Vector\ 3\ (Signed\ 16)$$

The type of the cpu as a whole can now be defined as ($Opcode$ will be defined later):

$$\begin{aligned} &cpu :: CpuState \\ &\qquad \rightarrow (Signed\ 16, Opcode, Vector\ 3\ (Index\ 5, Index\ 5)) \\ &\qquad \rightarrow (CpuState, Signed\ 16) \end{aligned}$$

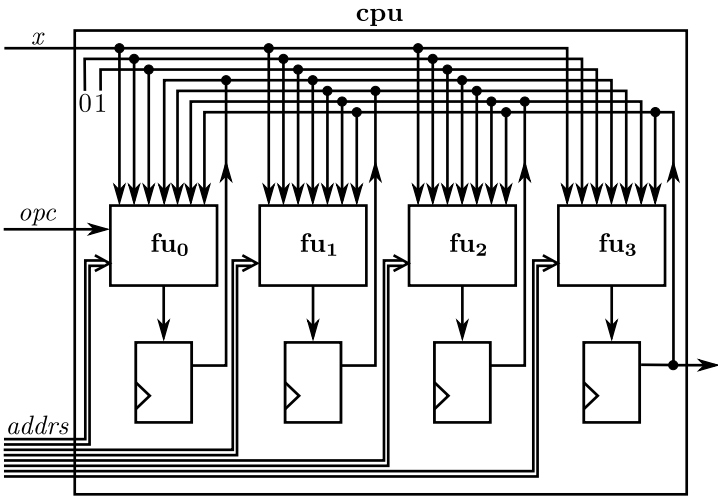

Fig. 1.6 Higher order cpu

Every function unit can be defined by the following higher order function, fu, which takes three arguments: the operation *op* that the function unit should perform, the six inputs, and the address pair (a_0, a_1). It selects two inputs, based on these addresses, and applies the given operation to them, returning the result ("!" is the operation for vector-indexing):

$$fu\ op\ inputs\ (a_0, a_1) = op\ (inputs!a_0)\ (inputs!a_1)$$

Exploiting partial application we now define (assuming that the binary functions *add* and *mul* already exist):

$$fun\ 1 = fu\ add$$
$$fun\ 2 = fu\ mul$$

Note that the types of these functions can be derived from the type of the cpu function and their usage below, thus determining what component instantiations are needed. For example, the function *add* should take two $Signed\ 16$ values and also deliver a $Signed\ 16$ value.

In order to define $fun\ 0$, the type *Opcode* and the function *multiop* that chooses a specific operation given the opcode, are defined first. It is assumed that the binary functions $shift$ (where $shift\ a\ b$ shifts a by the number of bits indicated by b) and *xor* (for the bitwise *xor*) exist.

$$\textbf{data}\ Opcode = Shift \mid Xor \mid Equal$$

$$\begin{aligned} multiop\ Shift &= shift \\ multiop\ Xor &= xor \\ multiop\ Equal &= \backslash a\ b \rightarrow if\ a == b\ then\ 1\ else\ 0 \end{aligned}$$

Note that the result of *multiop* is a binary function from two $Signed\ 16$ values into one $Signed\ 16$ value (hence, the **if-then-else** is needed since $a == b$ is a boolean). The type of *multiop* can be derived by the Haskell type system from the context.

The definition of $fun\ 0$, which takes an opcode as additional argument, is:

$$fun\ 0\ c = fu\ (multiop\ c)$$

The complete definition of the function *cpu* now is (note that $addrs$ contains three address *pairs*):

```
cpu s (x, opc, addrs) = (s', out)
  where
    inputs = x+ > (0+ > (1+ > s))
    s'     = V [ fun 0 opc inputs (addrs!0)
               , fun 1     inputs (addrs!1)
               , fun 2     inputs (addrs!2)
               ]
    out    = last s
```

Due to space restrictions, Figure 1.6 does not show the internals of each function unit. We remark that CλaSH generates e.g. *multiop* as a subcomponent of *fun* 0.

Remarks

In this example it is shown that also *user defined* higher order functions can be compiled by CλaSH, in this case the function fu. Note that in using this function, one may also exploit *partial application*, as in the definitions of $fun\ 0$, $fun\ 1$, $fun\ 2$.

In this example it is also shown that the designer may define his own enumeration types. As a final feature of CλaSH shown in this example we mention pattern matching: the function *multiop* is defined by pattern matching on the values of the type *Opcode*.

1.3.4 Floating point reduction circuit

The final example is a reduction circuit in which sequences of floating point numbers are added. Numbers come in one per clock cycle, sequence after sequence. When a sequence is finished, no further numbers belonging to that sequence will arrive.

We assume a pipelined floating point adder which we will exploit as optimally as possible, numbers belonging to different sequences may be in the pipeline at the same time. Only numbers belonging to the same sequence should be added together, so in order to keep numbers belonging to different sequences separated, they are labelled. This algorithm is introduced in [3] where it is also proven that numbers indeed may come in one per clock cycle without causing buffers to overflow.

The example shows that CλaSH can deal with architectures which consist of several components, where each component has its own state and is defined as a separate function.

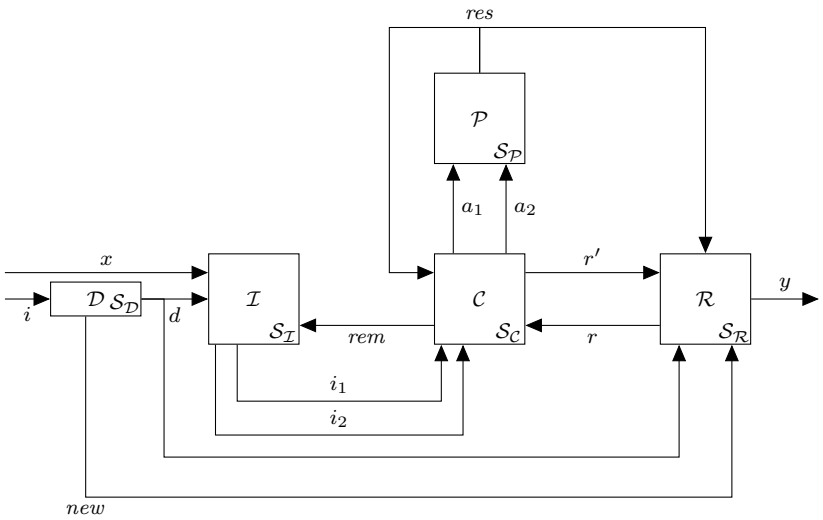

Fig. 1.7 Reduction circuit

The input (x, i) (see Figure 1.7) consists of a number and its row index. Since there will only be a limited number of rows "active" in the system, a limited number of labels is needed to distinguish different rows from each other. The discriminator component *discr* transforms the row index i into such a reduced label d after which the pair (x, d) enters the input component *inp* (which has a fifo ι as internal state). The boolean signal new_d says whether a new row starts (hence, the discriminator needs internal memory δ), and is used by the partial result buffer *res* to decide whether position d may be re-used for intermediate results of this new row. Both the memory ϱ in *res* and the number of labels used are big enough to be sure that the row which had label d before is ready at the moment d is re-used. Finally, the pipelined floating point adder *adder* (with internal state π) takes two numbers a_0, a_1 and outputs their sum several clock cycles later. Note that the pipeline π need not be completely full, so a value s delivered by *adder* may be undefined.

The central controller *contr* gathers the output s from *adder*, the corresponding partial result r from *res* (or an undefined value in case there is no corresponding previous result for the same row), and the first two elements i_0, i_1 from *inp* (without going into detail we remark that i_0 is always valid, whereas i_1 may be undefined). Based on these inputs, *contr* decides which values a_0, a_1 will be input into *adder*, which value r' will be given back to *res*, and the number of values *rem* that will be used from *inp* (and thus have to be removed from ι). This is done according to the following rules (in order of priority):

1. when s and the corresponding result r are both defined, then s and r together enter *adder*,
2. when s and the first element i_0 from *inp* have the same label, then s and i_0 enter *adder*,
3. when i_0, i_1 are both defined and their labels are the same, then i_0 and i_1 enter *adder*,
4. when i_0, i_1 are both defined but their labels are different, then i_0 and 0 enter *adder*,
5. when none of the above applies, no number enters *adder*.

In addition, when a number s with label d comes out of *adde* but s will not re-enter *adder*, s will be given to res for later use. Remember that every clock cycle a new value x enters *inp*.

In the context of this paper we will only show the definitions of the controller *contr* and of the full reduction circuit $reducer$. As seen above, there are *valid* values, consisting of a number and a label, and there are *invalid* values. We define the type $RValue$ for these values, containing a number of type *Float* and a label of type *Index* 127:

$$\textbf{data}\ RValue\ =\ Valid\ Float\ (Index\ 4)\ |\ NotValid$$

Three functions are needed to deal with such values, defined as follows:

$$\begin{array}{lll} value\ (Valid\ x\ d) & = & x \\ lbl\quad (Valid\ x\ d) & = & d \\ valid\ a & = & a/ = NotValid \end{array}$$

The definition of the controller *contr* can now be formulated as follows (nv and $zero$ are shorthand for $NotValid$ and $Valid$ 0 0, respectively):

$$\begin{array}{l} contr\ \gamma\ (i_0, i_1, s, r) = (\gamma, (a_0, a_1, rem, r')) \\ \quad \textbf{where} \\ \quad\quad (a_0, a_1, rem, r') \quad \begin{array}{ll} \mid valid\ s\ \&\&\ valid\ r & = (s, r, 0, nv) \\ \mid valid\ s\ \&\&\ lbl\ s == lbl\ i_0 & = (s, i_0, 1, nv) \\ \mid valid\ i_1\ \&\&\ lbl\ i_0 == lbl\ i_1 & = (i_0, i_1, 2, s) \\ \mid valid\ i_1 & = (i_0, zero, 1, s) \\ \mid \textbf{otherwise} & = (nv, nv, 0, s) \end{array} \end{array}$$

Note that the state parameter γ does not change, i.e., γ is empty. It is only there to match the required global structure of the definition.

The guards (indicated by "|", meaning "under the condition that") in this definition express the rules given above. Note that pattern matching is exploited in the *where*-clause: values are given to the four elements (a_0, a_1, rem, r') at the same time.

The definition of the full reduction circuit now looks as follows:

$$
\begin{aligned}
&\mathit{reducer}\ (\delta, \iota, \pi, \varrho, \gamma)\ (x, i) = ((\delta'', \iota'', \pi'', \varrho'', \gamma''), \mathit{out}) \\
&\quad \textbf{where} \\
&\qquad (\delta'', (\mathit{new}_d, d)) &&= \mathit{discr}\ \delta\ i \\
&\qquad (\iota'', (i_0, i_1)) &&= \mathit{inp}\ \iota\ (d, x, \mathit{rem}) \\
&\qquad (\pi'', s) &&= \mathit{adder}\ \pi\ (a_0, a_1) \\
&\qquad (\varrho'', (r, \mathit{out})) &&= \mathit{res}\ \varrho\ (d, \mathit{new}_d, s, r') \\
&\qquad (\gamma'', (a_0, a_1, \mathit{rem}, r')) &&= \mathit{contr}\ \gamma\ (i_0, i_1, s, r)
\end{aligned}
$$

Note that loops shown in the picture correspond to loops in the code, for example, a_0 is a result of *contr* and an argument for *adder*. At the same time, s is a result of *adder* and an argument for *contr*. In hardware there is no problem since these values come from memory elements. Also in the Haskell simulation there is no problem because of lazy evaluation.

Remarks

This example shows that *guards* can be dealt with by CλaSH. It also shows how to combine several components of an architecture together. However, to make the simulation run and to let GHC do its job properly, for now we have to mention the states of nested components in the signature of the combining component.

This reduction circuit was also written and hand-optimized in VHDL by the authors of [3]. Both the VHDL and the functional specification made the same global design decisions and local optimizations. Though it is difficult to compare the exact details of both specifications, the results of synthesizing both were very close: clock speed (around 170 MHz) and area (around 4500 CLB slices & LUTs) were within 10% of each other.

1.4 Conclusions and future research

At the moment CλaSH is a working prototype which is able to translate all the above examples into synthesizable VHDL. Work on several extensions is in progress, such as adding (limited) recursion, dealing with multi-clock domains, adding asynchronicity.

Also the formalism itself is topic of research, e.g., concerning formal properties of the reduction mechanism (confluence, termination), and its suitability for transformational design and for proving equivalence of specifications.

Though preliminary results are promising, further experiments have to be performed concerning a comparison with other HDL's on topics such as designer effort, readability and conciseness of code, as well as properties of synthesized hardware such as clock speed, area, longest path, etc.

References

1. C.P.R. Baaij, *CλaSH – From Haskell to Hardware*, Master Thesis, University of Twente, 2009.
2. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, Lava: hardware design in Haskell, in: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, New York, USA, 1998, pp. 174-184.
3. M.E.T. Gerards, J. Kuper, A.B.J. Kokkeler, E. Molenkamp, Streaming Reduction Circuit, in: *Proceedings of the 12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Patras, Greece, 2009, pp. 287–292
4. M. Kooijman, *Haskell as a Higher Order Structural Hardware Description Language*, Master Thesis, University of Twente, 2009.
5. J. Matthews, B. Cook, and J. Launchbury, Microprocessor specification in Hawk, in: *Proceedings of 1998 International Conference on Computer Languages*, 1998, pp. 90-101.
6. I. Sander, A. Jantsch, System Modeling and Transformational Design Refinement in ForSyDe, in: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2004, vol. 23, no. 1, pp. 17-32.
7. R. Sharp and O. Rasmussen, Using a language of functions and relations for VLSI specification, in: *FPCA 95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, New York, NY, USA, 1995, pp. 45-54.
8. M. Sheeran, μFP, a language for VLSI design, in: *LFP 84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, New York, NY, USA, 1984, pp. 104-112.